Applying Formal Methods to a Certifiably Secure Software System

Constance L. Heitmeyer, *Member*, *IEEE*, Myla M. Archer, *Member*, *IEEE Computer Society*, Elizabeth I. Leonard, *Member*, *IEEE Computer Society*, and John D. McLean

Abstract—A major problem in verifying the security of code is that the code's large size makes it much too costly to verify in its entirety. This paper describes a novel and practical approach to verifying the security of code which substantially reduces the cost of verification. In this approach, a compact security model containing only information needed to reason about the security properties of interest is constructed and the security properties are represented formally in terms of the model. To reduce the cost of verification, the code to be verified is partitioned into three categories and only the first category, which is less than 10 percent of the code in our application, requires formal verification. The proof of the other two categories is relatively trivial. Our approach was developed to support a Common Criteria evaluation of the separation kernel of an embedded software system. This paper describes 1) our techniques and theory for verifying the kernel code and 2) the artifacts produced, that is, a Top-Level Specification (TLS), a formal statement of the security property, a mechanized proof that the TLS satisfies the property, the partitioning of the code, and a demonstration that the code conforms to the TLS. This paper also presents the formal basis for the argument that the kernel code conforms to the TLS and consequently satisfies the security property.

Index Terms—Security, verification, specification, security kernels, tools, formal methods, software, software verification.

1 Introduction

A critical objective of many military systems is to protect the confidentiality and integrity of sensitive information. Preventing unauthorized disclosure and modification of sensitive information is of enormous importance in military systems since violations can jeopardize national security. Compelling evidence that military systems satisfy their security requirements is therefore required. A promising approach to demonstrating the security of code is formal verification, which has been successfully applied to algorithms such as floating-point division [1] and clock synchronization [2] and security protocols such as cryptographic protocols [3], [4]. However, most previous efforts to verify security-critical software have been extremely expensive. One reason is that these efforts often built security models containing too much detail (see, for example, [5]) or tried to prove too many properties (see, for example, [6]). The result was that model building and property proving were prohibitively expensive.

A challenging problem therefore is how to make the verification of security-critical code affordable. This paper describes a novel and practical approach to verifying the security of software that significantly reduces the cost of verification. This approach was formulated to support a Common Criteria evaluation of the security of a software-based embedded device called ED (Embedded Device). Satisfying the Common Criteria required a formal proof of correspondence between a formal specification of ED's security functions and its required security properties *and* a demonstration that ED's implementation satisfied the formal specification. ED, which processes data stored in different partitions of its memory, must enforce a critical security property called *data separation* to ensure, for example, that data in one memory partition neither influences nor is influenced by data in another partition. To guarantee that data separation is not violated (or, if it is violated, an exception occurs), ED relies on a separation kernel [7], a tamper-proof nonbypassable program mediating every access to memory.

The task of our group was to provide evidence to the certifying authority that the ED separation kernel enforces data separation. The kernel code, which contains on the order of 3,000 lines of C and assembly code, was annotated with preconditions and postconditions in the style of Hoare and Floyd. To provide evidence that ED enforces data separation, we produced a Top-Level Specification (TLS) of the separation-relevant behavior of the kernel, a formal statement of data separation, and a mechanized formal proof that the TLS satisfies data separation. Then, the annotated code was partitioned into three categories, each requiring a different proof strategy. Finally, the formal correspondence between the annotated code and the TLS was established. Five artifacts—the TLS, the formal statement of data separation, proofs that the TLS satisfies data separation, the organization of the annotated code into the three categories, and the documents showing correspondence of the code to the TLS—were presented, along with the annotated code, as evidence supporting the certification of ED.

This paper summarizes the process that we followed in producing evidence for the Common Criteria evaluation, describes the artifacts developed during the process, and presents the formal argument justifying our approach to

• *The authors are with the Information Technology Division, Naval Research Laboratory, 4555 Overlook Avenue, S.W., Washington, DC 20375. E-mail: {heitmeyer, archer, leonard, mclean}@itd.nrl.navy.mil.*

Manuscript received 24 Feb. 2007; revised 8 Aug. 2007; accepted 29 Aug. 2007; published online 27 Nov. 2007.
Recommended for acceptance by P. McDaniel and B. Nuseibeh.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSESI-0083-0207.
Digital Object Identifier no. 10.1109/TSE.2007.70772.

establishing conformance of the code with the TLS. The paper makes two major contributions. First, it describes a novel technique for partitioning the code into three different categories and for reasoning about the security of each category. This technique dramatically reduces the cost of verification (see Section 3.4). Second, it describes an original and practical method for demonstrating the security of code. Although the method combines a number of well-known techniques for specifying and reasoning about security (for example, a state machine model, an access control matrix [8], mechanized reasoning using PVS [9], and a demonstration of correspondence between the TLS and the annotated code), which techniques should be applied, how they can be applied, and how they can be combined were far from obvious and required significant discussion during the course of the project. Along the way, many alternative approaches and techniques were considered and several were discarded. What is notable about our effort is that, unlike many other efforts applying formal methods, ours was not a case study but a successful application of our formal techniques in the certification of a real-world system. Both our technique for partitioning the code and our method for proving the security of the code should prove cost-effective in future efforts to verify the security of software.

This article is organized as follows: Section 2 reviews the notion of a separation kernel, summarizes the requirements of a Common Criteria evaluation, and presents some details of ED. Section 3 describes the process that we followed to demonstrate data separation and describes the five artifacts that the process produced, including the three categories of code and the demonstration that each category of code is secure. Section 4 adapts the classical theory of refinement [10] to the proof that a concrete state machine model conforms to an abstract state machine model, thus providing a formal basis for proving the security of the kernel. Section 5 discusses the general class of security properties to which our techniques apply, that is, the class of properties that are preserved under refinement. It has been shown that safety properties belong to this class of properties [10]. Section 5 also discusses how our approach could be used to prove additional properties of the kernel. Sections 6 and 7 describe some lessons learned and four topics requiring more research, for example, the need for more powerful tool support. Section 8 discusses related work. Finally, Section 9 presents some conclusions.

2 Background

2.1 Separation Kernel

A *separation kernel* [7] mimics the separation of a system into a set of independent virtual machines by dividing the memory into partitions and restricting the information flow between those partitions. Separation kernels are being developed by commercial companies such as Wind River Systems, Green Hills Software, and LynuxWorks for military applications requiring Multiple Independent Levels of Security (MILS) [11]. In a MILS environment, a separation kernel acts as a reference monitor [12]: It is nonbypassable, evaluatable, always invoked, and tamper-proof.

2.2 Common Criteria

A number of international organizations established the Common Criteria to provide a single basis for evaluating the security of information technology products [13]. Associated with the Common Criteria are seven Evaluation Assurance Levels. EAL7, the highest assurance level, requires a formal specification of a product's security functions and its security model and formal proof of correspondence between the two.

2.3 Embedded Device

The device of interest, ED, processes data in an embedded system whose memory has been divided into nonoverlapping partitions. Although, at any given time, the data stored and processed by ED in one memory partition is classified at a single security level, ED may later reconfigure that partition to store and process data at a different security level. Because it stores and processes data classified at different security levels, security violations by ED could cause significant damage. To prevent violations of data separation, for example, the "leaking" of data from one memory partition to another, the ED design uses a separation kernel to mediate access to memory. By mediating every access, the kernel ensures that every memory access is authorized and that every transfer of data from one ED memory location to another is authorized. Any attempted memory access by ED that is unauthorized will cause an exception.

3 Code Verification Process

Given 1) source code annotated with Floyd-Hoare preconditions and postconditions and 2) a security property of interest, the problem is how to establish that the code satisfies the property. This section presents a five-step process for establishing the property, each step producing one of the five artifacts. The five steps of the process are listed as follows:

1. Formulate a TLS of the code as a state machine model in the style of [14], [15].
2. Formally express the security property as a property of the state machine model. Confirm that the property is preserved under refinement.
3. Translate the TLS and the property into the language of a mechanical prover and prove formally that the TLS satisfies the property.
4. Given source code annotated with preconditions and postconditions, partition the code into three categories—Event, Other, and Trusted Code—based on some criterion determined by the property of interest.
5. To demonstrate that the Event Code does not violate the property of interest, construct a) a mapping from the Event Code to the TLS events and from the code states to the states in the TLS and b) a mapping from the preconditions and postconditions of the TLS events to the preconditions and postconditions that annotate the corresponding Event Code. Demonstrate separately that Trusted Code and Other Code

are benign. Based on these results, conclude that the code refines the TLS.

Sections 3.1-3.5 describe how the above process was applied to the annotated code which implements ED's separation kernel. Each section describes, in turn, one of the five artifacts produced for ED. For the security property of interest in ED, that is, data separation, the criterion for partitioning was whether the code touched certain Memory Areas of Interest (MAIs). Event Code corresponds to events in the TLS that touch a MAI, Trusted Code touches a MAI but is not Event Code, and Other Code is neither Event Code nor Trusted Code. Section 3.4 precisely defines each of the three code categories. To provide evidence for ED's certification, a logician manually annotated the kernel code with assertions (that is, preconditions and postconditions) as a basis for validating the functional correctness of the code. During the certification process, evaluators from the certifying authority conducted a complete code walkthrough of the annotated code to check the correctness of the assertions. Checking that the kernel code enforces data separation uses these assertions in Step 5.

3.1 Top Level Specification

Major goals of the TLS are to provide a precise yet understandable description of the allowed security-relevant external behavior of ED's separation kernel and to make the assumptions on which the TLS is based explicit.[1] To achieve this, the TLS of the kernel behavior is represented in precise natural language as a state machine model by using the style of the Military Message System (MMS) security model [14], [15]. The advantage of precise natural language is that it enables stakeholders with differing backgrounds and objectives, that is, the project manager, software developers, evaluators, and the formal methods team, to communicate precisely about the required kernel behavior and helps ensure, early in the verification process, that misunderstandings are weeded out and issues are resolved. Another goal of the TLS is to provide a formal context and precise vocabulary for defining data separation.

Like the secure MMS model, the state machine representing the kernel behavior is defined in terms of an input alphabet, a set of states, an initial state, and a transform relation describing the allowed state transitions. The input alphabet contains internal and external events, where an *internal event* can cause the kernel to invoke some process, and an *external event* is performed by an external host. The transform (also called the *next-state relation*) is defined on triples consisting of an event in the input alphabet, the current state, and the new state. This section contains excerpts from the TLS. To provide intuition about the observable kernel behavior of ED, it also describes the five internal events and the single external event (the last event), listed in the leftmost column of Table 1.

Partitions, state variables, events, and states. We assume the existence of $n \geq 1$ dedicated memory partitions and a single shared memory area. We also assume the existence of the following sets:

1. For example, the assumptions make explicit those routines that the certification authority agreed were outside the scope of the formal verification.

TABLE 1
Excerpts from the Nonnull Portion of Access Control Matrix AM for Partition i, $1 \leq i \leq n$

Event e in H	Memory Areas in $\mathcal{M}$				
	B_i^1	D_i^1	D_i^2	...	G
Begin_Partition_i	—	—	—	—	—
Copy_B1In_D1In_i	R	W	—	—	—
Clear_D1_i	—	W	—	—	—
End_Partition_i	—	—	—	—	—
Other_NonPartProc	—	—	—	—	RW
...	...	...	...	...	...
ExtEv_B1In_i	RW	—	—	—	—

- V is a union of types, where each type is a nonempty set of values.
- R is a set of state variable names. For all r in R, $\mathrm{TY}(r) \subseteq V$ is the set of possible values of state variable r.
- $\mathcal{M}$ is a union of N nonoverlapping memory areas, each represented by a state variable.
- $H = P \cup E$ is a set of M events, where each event is either an internal event in P or an external event in E.

A *system state* is a function mapping each state variable name r in R to a value. Formally, for all $r \in R$, $s(r) \in \mathrm{TY}(r)$. Given state s and state variable r, we abbreviate $s(r)$ by r_s.

Memory areas. The N memory areas contain $N-1$ MAIs, where $N - 1 = mn$ and m is the number of MAIs per partition. Informally, a MAI is a memory area containing data whose leakage would violate data separation. The m MAIs for a partition i, $1 \leq i \leq n$, include partition i's input and output buffers and k data areas where data in partition i are stored and processed. The Nth memory area, called G, is the single shared memory area and contains all programs and data not residing in any MAI. The set $\mathcal{M}$ of all memory areas is defined as the union $A \cup \{G\}$, where $A = \{A_{i,j} \mid 1 \leq i \leq n \wedge\ 1 \leq j \leq m\}$ contains the mn MAIs. For all i, $1 \leq i \leq n$, $A_i = \{A_{i,j} \mid 1 \leq j \leq m\}$ is the set of memory areas for partition i. To ensure that they are nonoverlapping, the memory areas of $\mathcal{M}$ are required to be pairwise disjoint.

State variables. The set of state variables[2] contained in R are

- a partition id c,
- the N memory areas in $\mathcal{M}$, and
- a set of n sanitization vectors $\mathcal{W}[1], \ldots, \mathcal{W}[n]$, each vector containing k elements.

The partition id c is 0 if no data processing in any partition is in progress and it is i, $1 \leq i \leq n$, if data processing is in progress in partition i. (Data processing can occur in only one partition at a time.) For $1 \leq j \leq k$, the Boolean value of the jth element $\mathcal{W}^j[i]$ of the sanitization vector for partition i is *true* initially and if the jth memory area of the ith partition has been sanitized since it was last written, and otherwise *false*. A sanitized memory area is modeled as having the value 0.

Events. The set of internal events $P \subset H$ is the union of n sets, $P_1, \ldots, P_n$, of *partition events*, one set for each

2. By convention, state variable names may refer to the values of the variables.

partition i, and a singleton set Q. Thus, P is defined by $P = [\cup_{i=1}^{n} P_i] \cup Q$. Processing occurs on partition i when a sequence of events from P_i is processed. The first four events listed in Table 1 are partition events in some P_i. The first event, `Begin_Partition_`i, initiates data processing in partition i. The next two events process data stored in i's memory areas: Event `Copy_B1In_D1_`i copies data from B_i^1, which is an input buffer assigned to i, into a memory area D_i^1 of i and event `Clear_D1_`i sanitizes memory area D_i^1. The event `End_Partition_`i concludes data processing in partition i. Q's sole member is `Other_NonPartProc`, which is the fifth event listed in Table 1, an abstract event representing all internal events that invoke data processing in the shared memory area G. An example is the event that copies a shared algorithm, written by some external host into a shared input buffer, to some other part of G.

The set of external events $E \subset H$ is defined by $E = E^{\text{In}} \cup E^{\text{Out}} \cup \{\texttt{Ext\_Ev\_Other}\}$, where $E^{\text{In}} = \cup_{i=1}^{n} E_i^{\text{In}}$ and $E^{\text{Out}} = \cup_{i=1}^{n} E_i^{\text{Out}}$. E_i^{In} is the set of external events writing into or clearing the input buffers of partition i and E_i^{Out} is the set of external events reading from or clearing the output buffers of partition i. The event `Ext_Ev_Other` represents all other external events. `ExtEv_B1In_`i, the last event listed in Table 1, is an example of an external event in E^{In} which occurs when an external host writes data (to be processed in partition i) into the input buffer B_i^1.

Partition and nonpartition functions. Operations on data in partition i, for example, an operation copying data from one MAI in partition i to another MAI in i, are called *partition functions*. For all i, $1 \leq i \leq n$, and, for each internal event e in P_i, there exists a *partition function* Γ_e associated with e. For all $e \in P_i$, Γ_e has the signature $\Gamma_e : \text{TY}(a_1) \rightarrow \text{TY}(a_2)$, where a_1 and a_2 are MAIs in A_i. Thus, each function Γ_e, where e is an internal event in P_i, takes a single argument, that is, the value stored in some MAI a_1 and uses that argument to compute a value to be stored in MAI a_2 as the result of event e. A *nonpartition function* Γ_e has access to data in G only.

Access control matrix. Associated with the M events and N memory areas is an M by N access control matrix AM, which indicates the access privileges that each internal event e in P (and its associated process) and each external event e in H has for each memory area a in $\mathcal{M}$. The access privileges are either `null` for no access, `R` for read access, `W` for write access, or `RW` for both read and write access. Table 1 shows excerpts from the access control matrix AM. The leftmost column of Table 1 lists the events in H and the headings of the remaining columns list memory areas in $\mathcal{M}$. The rightmost column heading contains G, the only non-MAI, while the remaining column headings contain all MAIs for partition i. For all i, $1 \leq i \leq n$, AM shows the access privileges that each internal and external event has for each of i's memory areas and for memory area G. In Table 1, "$-$" denotes `null` access. For all i, j, $1 \leq i, j \leq n$, $i \neq j$, the access privilege that an event associated with i has to a memory area associated with j (not shown in Table 1) is `null`. Similarly, the access privilege that an event associated with j (not shown in Table 1) has to a memory area associated with i is also `null`.

To illustrate how AM limits access to the memory areas in $\mathcal{M}$, we consider the event in the second row of Table 1, that is, $e =$ `Copy_B1In_D1In_`i. Table 1 shows that a process invoked by e has read access to B_i^1, one of i's input buffers, and write access to D_i^1, one of i's data areas, and null access to all other memory areas in $\mathcal{M}$. Thus, for event e, $\text{AM}[e, B_i^1] = \texttt{R}$, $\text{AM}[e, D_i^1] = \texttt{W}$, and $\text{AM}[e, a] = \texttt{null}$ for all a, $a \in \mathcal{M}$, $a \notin \{B_i^1, D_i^1\}$. Similarly, the event `Clear_D1_`i can only write to D_i^1 and the abstract event `Other_NonPartProc` only has read and write access to G. The events that begin and end data processing on i, `Begin_Partition_`i and `End_Partition_`i, cannot write to any memory area. Finally, the external event `ExtEv_B1In_`i invokes a process that can only read and write into the input buffer B_i^1.

System. A system is a state machine whose transitions from one state to the next are triggered by events. Formally, a system Σ is a 4-tuple $\Sigma = (H, S, s_0, T)$, where

- H is the set of events,
- S is the set of states,
- s_0 is the initial state, and
- T is the system transform, a partial function from $H \times S$ into S. T is partial because not all events are "enabled" to be executed in the current state.

Initial state. In the initial state s_0, the partition id c is 0; for all i, $1 \leq i \leq n$, the MAIs in A_i are 0; and each element of the sanitization vectors $\mathcal{W}[1] \ldots \mathcal{W}[n]$ is *true*. Hence, in the initial state, no processing in any partition is authorized, only a nonpartition process is authorized to execute, all MAIs are zero, and all data areas are known to be sanitized.

System transform. The transform T is defined in terms of a set $\mathcal{R}$ of transform rules $\mathcal{R} = \{\mathcal{R}_e \mid e \in H\}$, where each *transform rule* $\mathcal{R}_e$ describes how an event e transforms a current state into a new state. The number of rules is M, one rule for each of the M events in H. No rule requires access privileges other than those defined by the access control matrix AM. The notation s and s' represents the current state and the new state, respectively. When an internal or external event e does not affect the value of any state variable r, when the precondition is not satisfied, or when the event e is not enabled, the value of r does not change from state s to state s' and the state variable r retains its current value, that is, $r_s = r_{s'}$.

To denote that no state variable changes, except those explicitly named, we write $\text{NOC}_{\hat{R}}$ (NO Change, except to variables in $\hat{R}$), where $\hat{R} \subset R$. This notation also covers the case where the ith element of a sanitization vector changes, but no other vector elements change. For example, the postcondition $r_{s'} = x \wedge \text{NOC}_{\{r\}}$, where $x \in \text{TY}(r)$, is equivalent to $r_{s'} = x \wedge \forall \hat{r} \in R$, $\hat{r} \neq r$: $\hat{r}_{s'} = \hat{r}_s$.

Suppose that s is a state in S, e is an event in H, and R is the set of state variables. Let $\texttt{pre}_e$ be a state predicate associated with e such that $\texttt{pre}_e$ evaluates to *true* if e has the potential to occur in state s and *false* otherwise. In addition, let $\texttt{post}_e$ be a predicate associated with e such that $\texttt{post}_e(s, s')$ holds whenever e occurs in state s and s' is a possible poststate of s when event e occurs in state s. Formally, the transform rule $\mathcal{R}_e$ in $\mathcal{R}$ is defined by

$$\mathcal{R}_e : \texttt{pre}_e(s) \Rightarrow \texttt{post}_e(s, s').$$

Whenever the result state of every event e is deterministic (which is true in the TLS for ED), the assertion $\texttt{post}_e(s, s')$

defines the poststate $s' = T(e, s)$. To make T total on $H \times S$, the complete definition of T is written as

$$T(e,s) = \begin{cases} s', & \text{if } \texttt{pre}_e(s), \text{ where } \texttt{post}_e(s,s') \\ s, & \text{otherwise.} \end{cases}$$

In the above definition, $\texttt{pre}_e(s)$ is not satisfied implies that e has no effect, that is, essentially e does not occur. Abstractly, this models raising an exception and halting.

Examples of transform rules. For all i, $1 \le i \le n$, the transform rule for $e = \texttt{Begin\_Partition\_}i$, which begins data processing on i, is denoted $\mathcal{R}_{\text{Begin\_Partition\_}i}$. A precondition for event e is that the partition id is 0 (that is, the system is not currently processing data on any partition) and the postcondition for e is that the partition id is i. For all i, $1 \le i \le n$, and, for all states s and s', the rule $\mathcal{R}_e$ for $e = \texttt{Begin\_Partition\_}i$ is defined by

$$\mathcal{R}_{\text{Begin\_Partition\_}i}: \quad c_s = 0 \Rightarrow c_{s'} = i \wedge \text{NOC}_{\{c\}}.$$

The notation $\text{NOC}_{\{c\}}$ means that no state variable other than the partition id c can change. Similarly, for all i, $1 \le i \le n$, the rule $\mathcal{R}_e$ for $e = \texttt{End\_Partition\_}i$, which ends data processing on i, is defined by

$$\begin{aligned} \mathcal{R}_{\texttt{End\_Partition\_}i}: \quad & c_s = i \ \wedge \ \forall\, 1 \le j \le k, \mathcal{W}_s^j[i] = \textit{true} \\ \Rightarrow \quad & c_{s'} = 0 \ \wedge \ \text{NOC}_{\{c\}}. \end{aligned}$$

The expression "$\forall\, 1 \le j \le k, \mathcal{W}_s^j[i] = true$" in the above rule means that each element of the sanitization vector for i must be *true* for data processing on i to end. This can be achieved by invoking *clear events* such as $\texttt{Clear\_D1\_}i$ prior to invoking $\texttt{End\_Partition\_}i$. The purpose of this precondition is to ensure that all data areas of partition i are sanitized prior to processing on G, on partition j, $j \ne i$, or on a new configuration of i. The transform rules $\mathcal{R}_{\text{Begin\_Partition\_}i}$ and $\mathcal{R}_{\text{End\_Partition\_}i}$ are the only rules that change the value of the partition id c. Together, these rules constrain the partition id c to change from 0 to nonzero or from nonzero to 0.

Processing on a partition i can include copying data from an input buffer of partition i to a data area of partition i. Consider again the internal event $e = \texttt{Copy\_B1In\_D1In\_}i$, whose transform rule is denoted $\mathcal{R}_{\text{Copy\_B1In\_D1In\_}i}$. The preconditions for e are:

(1) The partition id c is equal to i.
(2) The invoked process must have read access R for partition i's Input Buffer 1 and write access W for Data Area 1 in partition i.

Postconditions for e are:

(3) The element for Data Area 1 in partition i's sanitization vector becomes *false* (because the event stores the value of Buffer 1 in Data Area 1).
(4) A function of the value in partition i's Input Buffer 1 is written into partition i's Data Area 1.
(5) No other state variable changes.

For all i, the rule $\mathcal{R}_e$ for event $e = \texttt{Copy\_B1In\_D1In\_}i$ is defined by

$$\begin{aligned} \mathcal{R}_{\texttt{Copy\_B1In\_D1In\_}i}: \ & c_s = i \ \wedge & (1)\\ & \text{AM}[e, B_i^1] = \texttt{R} \wedge \text{AM}[e, D_i^1] = \texttt{W} & (2)\\ \Rightarrow \ & \mathcal{W}_{s'}^1[i] = \textit{false} \ \wedge & (3)\\ & D_{i,s'}^1 = \Gamma_e(B_{i,s}^1) \ \wedge & (4)\\ & \text{NOC}_{\{\mathcal{W}^1[i], D_i^1\}}. & (5) \end{aligned}$$

As the fourth and final example of a transform rule, consider the rule for the internal event $e = \texttt{Other\_NonPartProc}$, which represents all nonpartition processing events. The precondition is that the partition id c is 0 (that is, the system is not currently processing data on any partition). The effect is that some part of memory area G may change. The rule $\mathcal{R}_e$ for $e = \texttt{Other\_NonPartProc}$ is defined by

$$\begin{aligned} \mathcal{R}_{\text{Other\_NonPartProc}}: \quad & c_s = 0 \ \wedge \ \text{AM}[e, G] = \texttt{RW} \\ \Rightarrow \quad & G_{s'} = \Gamma_e(G_s) \wedge \\ & \forall\, r \in R,\ r \ne G : r_{s'} = r_s. \end{aligned}$$

3.2 Security Property: Data Separation

To operate securely, ED must enforce data separation, that is, it must prevent insecure data flows. Informally, this means that ED must prevent data in a partition i from influencing or being influenced by 1) data in a partition j, where $i \ne j$, 2) data in an earlier configuration of partition i, or 3) data stored in G. To demonstrate that the TLS enforces data separation, we proved that it satisfies five subproperties, namely, *No-Exfiltration*, *No-Infiltration*, *Temporal Separation*, *Separation of Control*, and *Kernel Integrity*. Each subproperty is formally defined below by using the notation in Section 3.1.

3.2.1 No-Exfiltration Property

The No-Exfiltration Property states that data processing in any partition j cannot influence data stored outside the partition. This property is defined in terms of the set A_j (the MAIs of partition j); the entire memory $\mathcal{M}$; the internal events in P_j, which invoke data processing in j; and the external events in $E_j^{\text{In}} \cup E_j^{\text{Out}}$, which affect data in j's input and output buffers.

Property 3.1 (No-Exfiltration). *Suppose that states s and s' are in state set S, event e is in H, memory area a is in $\mathcal{M}$, and j is a partition, $1 \le j \le n$. Suppose further that $s' = T(e, s)$. If e is an event in $P_j \cup E_j^{\text{In}} \cup E_j^{\text{Out}}$ and $a_s \ne a_{s'}$, then a is in A_j.*

3.2.2 No-Infiltration Property

The No-Infiltration Property states that data processing in any partition i is not influenced by data outside that partition. It is defined in terms of the set A_i, which contains the MAIs of partition i.

Property 3.2 (No-Infiltration). *Suppose that states s_1, s_2, s_1', and s_2' are in S, event e is in H, and i is a partition, $1 \le i \le n$. Suppose further that $s_1' = T(e, s_1)$ and $s_2' = T(e, s_2)$. If, for all a in A_i, $a_{s_1} = a_{s_2}$, then, for all a in A_i, $a_{s_1'} = a_{s_2'}$.*

3.2.3 Temporal Separation Property

This property ensures that no data (for example, Top Secret data) stored in the ith partition during one configuration of

the partition can remain in any memory area of a later configuration (for example, processing Unclassified data) of that same partition i. The property is guaranteed if the k data areas in any partition i are clear when the system is not processing data in that partition, for example, from the end of a processing thread in one partition to the start of a new processing thread in the same or a different partition.[3] The set of states in which the system is not processing data stored in a partition is exactly the set of states in which the partition id c is 0. This fact is used in stating the property.

Property 3.3 (Temporal Separation). *For all states s in S, for all i, $1 \leq i \leq n$, if the partition id c_s is 0, then the k data areas of partition i are clear, that is, $D^1_{i,s} = 0, \ldots, D^k_{i,s} = 0$.*

3.2.4 Separation of Control Property

This property states that, when data processing is in progress on partition i, no data is being processed on partition j, $j \neq i$, until processing on partition i terminates. The property is defined in terms of the partition id c and the set D_i of k data areas in partition i, $D_i = \{D^j_i \mid 1 \leq j \leq k\}$.

Property 3.4 (Separation of Control). *Suppose that states s and s' are in S, event e is in H, data area a is in $\mathcal{M}$, and j, where $1 \leq j \leq n$, is a partition id. Suppose further that $s' = T(e, s)$. If neither c_s nor $c_{s'}$ is j, then $a_s = a_{s'}$ for all $a \in D_j$.*

3.2.5 Kernel Integrity Property

The Kernel Integrity Property states that, when data processing is in progress on partition i, the data stored on memory area G does not change. This property is defined in terms of G and the set P_i of events for partition i.

Property 3.5 (Kernel Integrity). *Suppose that states s and s' are in state set S, event e is in H, and i is a partition, $1 \leq i \leq n$. Suppose further that $s' = T(e, s)$. If e is a partition event in P_i, then $G_{s'} = G_s$.*

3.3 Formal Verification

To formally verify that the TLS enforces data separation, the natural language formulation of the TLS was translated into TAME (Timed Automata Modeling Environment) [16], [17], a front end to the mechanical prover PVS [18] which helps a user specify and reason formally about automata models. This translation requires the completion of a template to define the initial states, state transitions, events, and other attributes of the state machine Σ. The TAME specification provides a machine version of the TLS that can be shown mechanically to satisfy the properties defined in Section 3.2. After constructing the TAME specification of the TLS, we formulated two sets of TLS properties in TAME —invariant properties and other properties—which together formalize the five subproperties. Then, for each set of properties, we interactively constructed (TAME) proofs showing that the TAME specification satisfies each property. The scripts of these proofs, which are saved by PVS, can be rerun easily by the evaluators and serve as the formal proofs of data separation. One benefit of TAME is that the saved PVS proof scripts can be largely understood without rerunning them in PVS.

3. The proof of this property depends on the constraint noted earlier: If c changes, it changes from 0 to nonzero or vice versa.

3.4 Partitioning the Code

To show formally that the separation kernel enforces data separation, we must prove that the kernel is a secure partial instantiation of the state machine Σ defined by the TLS. The formal verification described in Section 3.3 establishes formally that a strict instantiation of the TLS enforces data separation. A *partial instantiation* of the TLS is an implementation that contains fine-grained details which do not correspond to the state machine Σ defined in the TLS. A *secure partial instantiation* of the TLS is a partial instantiation of the TLS in which the fine-grained details that do not correspond to the TLS are benign. Section 4 contains the formal foundation for the proof that the code is a secure partial instantiation of the TLS.

The proof that the code for the ED kernel is a secure partial instantiation of the TLS is based on a demonstration that all kernel code falls into three major categories and one subcategory, with proofs that the code in each category satisfies certain properties. The categories are given as follows:

1. *Event Code* is kernel code that implements a TLS internal event e in P and touches one or more MAIs. For each segment of Event Code, it is checked that
   i. the concrete translation of the precondition in the TLS for the corresponding event e is satisfied at the point in the kernel code where the execution of the event code is initiated, and
   ii. the concrete translation of the postcondition in the TLS for the corresponding event e is satisfied at the conclusion of Event Code execution.
2. *Trusted Code* is kernel code that touches MAIs but is not Event Code. This code does not correspond to behavior defined by the TLS and may have read and write access both to MAIs and to memory areas outside the MAIs. It is validated either by a proof that the code does not permit any nonsecure information flows or, in rare instances, by external certification. The TLS makes explicit any assumptions used in connection with the Trusted Code and its behavior. The proofs for a given segment of the Trusted Code characterize the entire functional behavior of that Trusted Code by using Floyd-Hoare style assertions at the code level and show that no nonsecure information flows can result from that code.
3. *Other Code* is the kernel code that is neither Event Code nor Trusted Code. More specifically, Other Code is kernel code which does not correspond to any behavior defined by the TLS and has no access to any MAI.
   a. A subset of the Other Code, called *Validated Code*, is code with no access to MAIs which is still security relevant because it performs functions necessary for the kernel to enforce data separation. These functions include setting up the MMU, establishing preconditions for the Event Code, etc. Floyd-Hoare style assertions at the code

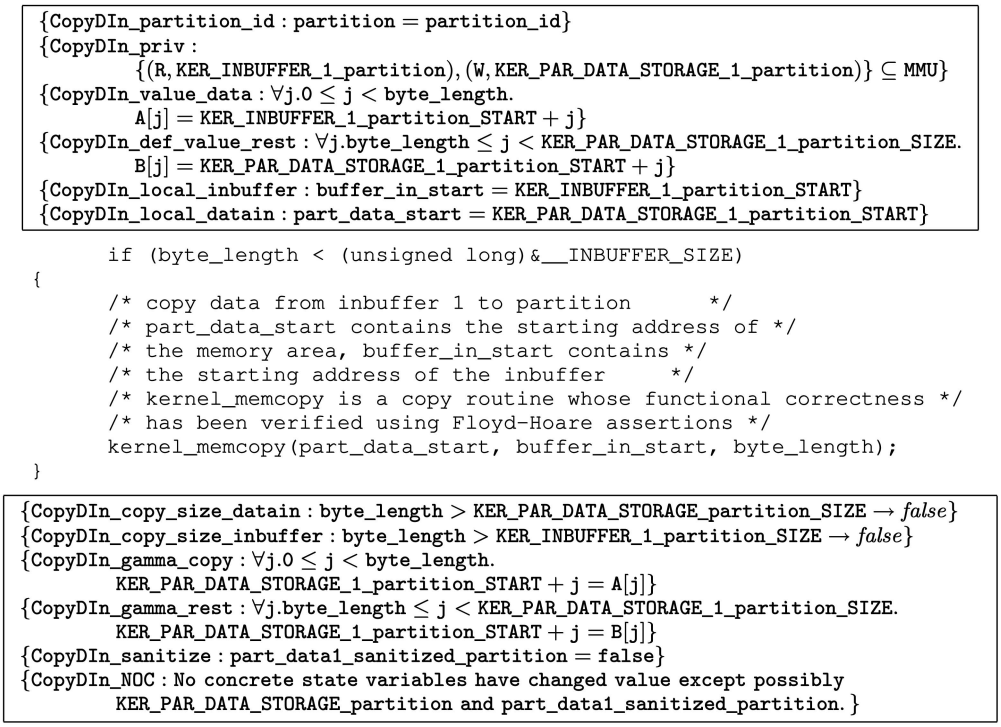

Fig. 1. Event Code and code-level assertions for the event Copy_B1In_D1In_i.

level are used to prove that Validated Code correctly implements the required functions.

The kernel code was manually partitioned into Event, Trusted, and Other Code. A first pass through the code showed that only a small number of functions could reset the MMU (that is, change the access permissions to memory areas). Apple's Xcode development tool [19] was used to search the kernel code for all calls to these functions. Each such call was inspected to determine the memory areas to which access was granted. By analyzing the access granted to code segments categorized as Other Code, one can verify that functions called in these code segments have no access to any MAI.

Partitioning the code in this manner dramatically reduces the cost of code verification since only the Event Code, a small part of the code, needs to be checked for conformance to the TLS. In ED, Event Code and Trusted Code comprised less than 10 percent of the code. The remaining 90 percent was Other Code.

3.5 Demonstrating Code Conformance

Demonstrating that the kernel code conforms to the TLS requires the definition of two mappings. To establish correspondence between concrete states in the code and abstract states in the TLS, a function α is defined which relates concrete states to abstract states by relating concrete entities (such as memory areas, code variables, and logical variables) to abstract state variables in the TLS (such as MAIs and the partition id) and mapping the value space of each concrete entity to that of its corresponding abstract state variable. For example, α maps the actual physical addresses of the MAIs to their corresponding abstract state variables in the TLS. In the ED kernel code, α maps a global variable `partition_id`, corresponding to the partition id, to the TLS partition id variable c. The TLS sanitization vectors have no analogs in the code. Instead, a predicate can be inferred from the code to indicate whether a memory area is sanitized. To represent sanitization in the concrete machine, new logical variables (for example, `part_data1_sanitized_`i) are introduced, and α maps these variables to elements of the sanitization vectors in the TLS. The map α also maps the Event Code to events in the TLS. Another map Φ relates assertions at the abstract TLS level to equivalent assertions at the code level derived from the abstract assertions and the map α. See Section 4 for more details.

Using Φ to relate preconditions and postconditions for an event in the TLS to the derived preconditions and postconditions for the corresponding Event Code, we next determine, for each piece of Event Code, sets of code-level preconditions and postconditions that match the derived preconditions and postconditions as closely as possible. Fig. 1 shows the Event Code corresponding to the internal event `Copy_B1In_D1In_`i in the TLS (see Section 3.1) and the code-level preconditions and postconditions for this Event Code. Although the Event Code for `Copy_B1In_D1In_`i consists of only a single function call, generally, Event Code may consist of any block of code. In Fig. 1, the top box contains the preconditions, then the indented Event Code is listed, and, finally, the bottom box contains the postconditions. Each precondition and postcondition has the form {`Assertion_Name : Assertion`}. Generally, the match between assertions in the TLS and derived code-level assertions is not exact because auxiliary assertions are added (see Fig. 1) 1) to express the correspondence between variables in the code and physical memory areas[4] (for example, `CopyDIn_local_datain`), 2) to save values in memory areas as the values of logical variables

4. This facilitates Floyd-Hoare reasoning at the code level.

TABLE 2
Mapping Preconditions in the Code to Preconditions in the TLS

Precondition $\Phi(\texttt{pre}_e)(s_c)$ Desired in the Code	Assertion in Annotated Code	Precondition $\texttt{pre}_e(s)$ in the TLS	Ref. No.	Description
CopyDIn_partition_id	§8.4,P5	$c_s = i$	(1)	Partition id is i
CopyDIn_priv	§8.4,TLS1*	$\mathrm{AM}(e, B_i^1) = \texttt{R}$ $\mathrm{AM}(e, D_i^1) = \texttt{W}$	(2)	R access for Input Buffer 1, W access for Data Area 1
CopyDIn_value_data	§8.4,P4*	$B_{i,s}^1$	-	Value of data in Input Buffer 1
CopyDIn_def_value_rest	§8.4,TLS4	$D_{i,s}^1$	-	Value of Data Area 1
CopyDIn_local_inbuffer	§8.4, TLS3*	-	-	Local variable for Input Buffer 1
CopyDIn_local_datain	§8.4,TLS2*	-	-	Local variable for Data Area 1

TABLE 3
Mapping Postconditions in the Code to Postconditions in the TLS

Postcondition $\Phi(\texttt{post}_e)(s_c, s_c')$ Desired in the Code	Assertion in Annotated Code	Postcondition $\texttt{post}_e(s, s')$ in the TLS	Ref. No.	Description
CopyDIn_copy_size_datain	§8.4,R2*	-	-	Wrong size → Error return
CopyDIn_copy_size_inbuffer	§8.4, R3*	-	-	Wrong size → Error return
CopyDIn_gamma_copy	§8.4, R7*	$D_{i,s'}^1 = \Gamma(B_{i,s}^1)$	(4)	Copy to Data Area 1
CopyDIn_gamma_rest	§8.4,TLS6	-		Rem Data Area 1 unchged
CopyDIn_sanitize	§8.4,TLS5*	$\mathcal{W}_{s'}^1[i] = false$	(3)	Data Area 1 not sanitized
CopyDIn_NOC	By inspection	$\mathrm{NOC}_{\{\mathcal{W}^1[i], D_i^1\}}$	(5)	No other change

(for example, CopyDIn_value_data), and 3) to express error conditions (for example, CopyDIn_copy_size_datain) that the TLS abstracts away via type correctness. The derivation of the necessary code-level assertions is also complicated by the code itself. For example, although there is a global variable partition_id in the code, in many of the routines implementing Event Code, the partition id used in the routine is an argument that is passed into the routine. This results in a code-level precondition asserting that the local variable for the partition id is equal to the global variable partition_id (for example, CopyDIn_partition_id in Fig. 1).

After defining the desired sets of code-level preconditions and postconditions, we check whether these assertions are among the assertions already proven in the annotated C code.[5] The annotated C code often refers to memory areas by indexing into arrays that define memory maps in the code, whereas the mapping α refers to memory areas by their actual physical addresses. Thus, to be equivalent to the desired assertions, the assertions in the annotated code frequently need dereferencing. For example, the annotated C code assertion §8.4, TLS2 (see Table 2) is defined by

```
part_data_start =
  (unsignedchar*)ker_rtime_mmu_map[partition].part_data_start,
```

which sets the variable part_data_start to the starting address of the data area in the partition by indexing into the real-time memory map in the code and selecting the part_data_start member of the structure corresponding to that array element. Dereferencing the index into the array and pointer into the structure yields the memory area KER_PAR_DATA_STORAGE_1_partition_START, the actual physical address of the partition data area, which stores the value used in the code-level precondition CopyDIn_local_datain (see the last line of the top box in Fig. 1).

In our initial attempt to match a precondition and postcondition in the annotated C code with each desired precondition and postcondition, either

- the desired assertion exactly matched an assertion in the annotated code,
- the desired assertion exactly matched an assertion in the annotated code, except dereferencing was required,
- the desired assertion was a close but not exact match of an assertion in the annotated code, or
- no code assertion exactly or approximately matched the desired assertion.

We worked with the logician who annotated the C code to ensure that assertions corresponding to all of the desired preconditions and postconditions were added to and verified on the code. (In general, it is sufficient to include strongest postconditions implying our derived assertions.) For example, assertions about a predicate SANITIZED on memory areas were added to the annotated code to provide correspondence to the necessary code-level assertions about the sanitization of memory areas. To show correspondence between the preconditions and postconditions in the code and the TLS, two tables were created for each TLS event. Tables 2 and 3 are the correspondence tables for the preconditions and postconditions of the transform rule for the TLS event Copy_B1In_D1In. In the tables, s and $s' = T(e, s)$ represent the abstract prestate and poststate, s_c and s_c' represent the concrete prestate and poststate, and Φ, which is formally defined in Section 4, maps abstract predicates to corresponding concrete predicates.

5. In general, the annotated code contains significantly more assertions than are needed to show correspondence because the annotations were primarily designed to show the functional correctness of the code.

In Tables 2 and 3, the first column contains the label of a desired code-level precondition or postcondition from Fig. 1, the second column gives the location (the section number and assertion label) of the corresponding assertion in the annotated C code, the third column contains the corresponding precondition or postcondition (if any) in the TLS, the fourth column gives the reference number of the corresponding assertion in the transform rule, and the fifth column briefly describes the assertion. In cases where no corresponding assertion exists in the TLS, "$-$" appears in both the third and fourth columns. An asterisk "$*$" in the second column indicates that, for equivalence between the assertion in the annotated code and the desired code assertion to hold, the assertion in the annotated code requires dereferencing.

Tables 2 and 3 show that, for every precondition and postcondition of `CopyB1In_D1In_`i, there is an equivalent precondition or postcondition in the annotated code. Therefore, we have shown that, for `CopyB1In_D1In_`i, the full code-level preconditions and postconditions imply the TLS preconditions and postconditions. Using the same techniques, we have also demonstrated the analogous result for the remaining events. As shown in Section 4, this demonstrates that the Event Code implementing the separation kernel is a refinement of the TLS.

4 Formal Foundations

Section 4.1 adapts the classical theory of *refinement* [10], a technique for proving that a concrete state machine model conforms to (that is, is a refinement of) an abstract state machine model, into a form that we can use to show that the behavior of the kernel code conforms to the behavior captured in the TLS. Section 4.1 also covers the formal foundation for our method of proving refinement and describes how we have applied it to verify that the kernel code correctly implements the TLS. The refinement proof technique that we use is closed under iteration (see Appendix A for the formal statement and proof). Section 4.2 compares our use of refinement relations and our method of verifying them with other techniques for applying and proving refinements.

4.1 Adapting the Classical Theory of Refinement

To begin, a function α is defined which maps each concrete state at the code level to a corresponding abstract state in the TLS state machine Σ by relating variables at the concrete code level to variables at the abstract TLS level. Variables at the concrete level include variables in the code, predicates defined on the code, logical history variables, and memory areas. Among the most important memory areas treated as concrete state variables are the data areas and the input and output buffers assigned to each partition, all of which are central to reasoning about possible information flows. Provided each possible value of a concrete state variable can be represented by some possible value of the corresponding abstract state variable (as is true for ED), the map α from concrete to abstract state variables induces a map $\alpha : S_c \rightarrow S_a$ from concrete to abstract states in the obvious way.[6] Once α is defined at the level of states in terms of state variables and their values, the set E_c of Event Code segments is identified, and α is extended to map each code segment e_c in E_c to a corresponding internal event $e_a = \alpha(e_c)$ in the TLS.[7]

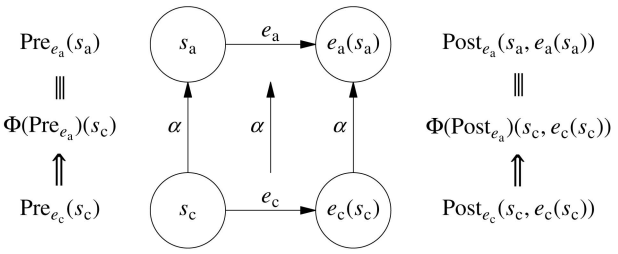

Fig. 2. Relations to establish between concrete and abstract transitions and preconditions and postconditions.

The map α from concrete states to abstract states provides a means of taking any predicate $P_a : S_a \rightarrow \mathit{Bool}$ on abstract states and deriving a corresponding predicate $\Phi(P_a) : S_c \rightarrow \mathit{Bool}$ on concrete states as follows:

$$\Phi(P_a)(s_c) \triangleq P_a(\alpha(s_c)),$$

where s_c is any state in S_c. Analogously, α can be used to derive a predicate $\Phi(P_a) : S_c \times S_c \rightarrow \mathit{Bool}$ on pairs of concrete states from a predicate on pairs of abstract states as follows:

$$\Phi(P_a)(s_c^1, s_c^2) \triangleq P_a(\alpha(s_c^1), \alpha(s_c^2)),$$

where s_c^1 and s_c^2 are any states in S_c. The map Φ is used to relate preconditions and postconditions in the code to preconditions and postconditions in the TLS (see Fig. 2). Note that preconditions (at both levels) apply only to one state. To capture the fact that an event changes only certain state variables (indicated at the abstract level by the notation `NOC`), the postconditions are represented at both levels as predicates on two states.

In Fig. 2, we follow the convention of representing $\alpha(s_c)$ by s_a. Note that, although the preconditions and postconditions on the concrete and abstract transitions in Fig. 2 are denoted analogously, their required relationships to their corresponding transitions differ. In particular, the precondition $\mathrm{Pre}_{e_a}(s_a)$ is a guard that, when false, prevents e_a from firing, while the precondition $\mathrm{Pre}_{e_c}(s_c)$ is simply an assertion known to hold before e_c fires. Moreover, the postcondition $\mathrm{Post}_{e_a}(s_a, e_a(s_a))$ is intended to capture the effect of the action e_a on the state s_a, while the postcondition $\mathrm{Post}_{e_c}(s_c, e_c(s_c))$ is simply an assertion known to hold for the states before and after e_c fires. Hence, the requirements for the abstract preconditions and postconditions fulfill the requirements for concrete preconditions and postconditions (but not vice versa). Thus, in our refinement proof method

6. To distinguish abstract from concrete entities, this section tags abstract entities with an a and concrete entities with a c. For example, S_a represents the abstract states and S_c represents the concrete states.

7. When external events, which have no corresponding event code, can occur in a system (as is true in the ED kernel), α must be similarly extended to map these "concrete" external events to their abstract equivalents. At both levels, the preconditions of external events are *true* and their postconditions capture their effects.

below, an abstract TLS can play a role analogous to concrete code with respect to a still more abstract TLS. For details, see Appendix B.

To establish equivalence between the behavior of the kernel code and a subset of the behavior modeled in the TLS, it is sufficient to prove, in the simplest case, that, for every $\mathrm{e_c}$ in $\mathrm{E_c}$, the following conditions hold:

1. Whenever the concrete code segment e_c is ready to execute in state $\mathrm{s_c}$, some concrete precondition $\mathrm{Pre}_{e_\mathrm{c}}$ holds, where $\mathrm{Pre}_{e_\mathrm{c}}$ implies $\Phi(\mathrm{Pre}_{e_\mathrm{a}})$, the concrete precondition derived from the abstract precondition for $e_\mathrm{a} = \alpha(e_\mathrm{c})$.
2. Whenever the concrete precondition $\mathrm{Pre}_{e_\mathrm{c}}$ holds for the current program state $\mathrm{s_c}$, some concrete postcondition $\mathrm{Post}_{e_\mathrm{c}}$ holds for the pair of program states $(s_\mathrm{c}, e_\mathrm{c}(s_\mathrm{c}))$ immediately before and immediately after the execution of e_c, where $\mathrm{Post}_{e_\mathrm{c}}$ implies $\Phi(\mathrm{Post}_{e_\mathrm{a}})$, the concrete postcondition derived from the abstract postcondition for e_a.
3. The diagram in Fig. 2 commutes whenever $\mathrm{Pre}_{e_\mathrm{c}}(s_\mathrm{c})$ holds.

Although this method requires the proof of conditions 1, 2, and 3, it is essentially condition 3 that is needed for α to be a refinement mapping. To prove condition 3, it is normally sufficient to prove conditions 1 and 2.

Theorem 4.1. *Provided* $\forall s, s' \in S_\mathrm{a} : \mathrm{Pre}_{e_\mathrm{a}}(s) \Rightarrow [\mathrm{Post}_{e_\mathrm{a}}(s, s') \equiv (s' = e_\mathrm{a}(s))]$ *(this holds for* `post`$_\mathrm{e}$ *in the TLS transform described in Section 3.1), conditions 1 and 2 imply condition 3.*

Proof. By hypothesis, we know that

(i) $\forall s, s' \in S_\mathrm{a} : \mathrm{Pre}_{e_\mathrm{a}}(s) \Rightarrow [\mathrm{Post}_{e_\mathrm{a}}(s, s') \equiv (s' = e_\mathrm{a}(s))]$

and may assume that conditions 1 and 2 hold. Further, by the hypothesis of condition 3, we may also assume that

(ii) $\mathrm{Pre}_{e_\mathrm{c}}(s_\mathrm{c})$.

By condition 1, it follows from (ii) that $\Phi(\mathrm{Pre}_{e_\mathrm{a}})(s_\mathrm{c})$, which means, by the definition of Φ, that

(iii) $\mathrm{Pre}_{e_\mathrm{a}}(\alpha(s_\mathrm{c}))$.

Furthermore, by condition 2, we have

(iv) $\mathrm{Pre}_{e_\mathrm{c}}(s_\mathrm{c}) \Rightarrow \mathrm{Post}_{e_\mathrm{c}}(s_\mathrm{c}, e_\mathrm{c}(s_\mathrm{c}))$,

and

(v) $\mathrm{Post}_{e_\mathrm{c}}(s_\mathrm{c}, e_\mathrm{c}(s_\mathrm{c})) \Rightarrow \Phi(\mathrm{Post}_{e_\mathrm{a}})(s_\mathrm{c}, e_\mathrm{c}(s_\mathrm{c}))$.

Thus,

	$\mathrm{Pre}_{e_\mathrm{c}}(s_\mathrm{c})$	(by (ii))
$\Rightarrow$	$\mathrm{Post}_{e_\mathrm{c}}(s_\mathrm{c}, e_\mathrm{c}(s_\mathrm{c}))$	(by (iv))
$\Rightarrow$	$\Phi(\mathrm{Post}_{e_\mathrm{a}})(s_\mathrm{c}, e_\mathrm{c}(s_\mathrm{c}))$	(by (v))
$\Leftrightarrow$	$\mathrm{Post}_{e_\mathrm{a}}(\alpha(s_\mathrm{c}), \alpha(e_\mathrm{c}(s_\mathrm{c})))$	(by the definition of Φ)
$\Leftrightarrow$	$\alpha(e_\mathrm{c}(s_\mathrm{c})) = e_\mathrm{a}(\alpha(s_\mathrm{c}))$	(by (i) and (iii)).

But, the last assertion means that the diagram in Fig. 2 commutes, which is the conclusion of condition 3. □

The hypothesis of Theorem 4.1 does not truly limit its use, provided that the abstract postcondition exactly captures all possible effects of the abstract transition. In particular, suppose that the definition of the abstract transition allows nondeterminism, and that one has established, based on conditions 1 and 2 and the hypothesis of condition 3, that $\mathrm{Post}_{e_\mathrm{a}}(\alpha(s_\mathrm{c}), \alpha(e_\mathrm{c}(s_\mathrm{c})))$, that is, that $\mathrm{Post}_{e_\mathrm{a}}(s_\mathrm{a}, \alpha(e_\mathrm{c}(s_\mathrm{c})))$. Then, to fulfill the hypothesis of Theorem 4.1, one can simply replace e_a by its deterministic instance for which the abstract poststate $e_a(s_\mathrm{a})$ is $\alpha(e_\mathrm{c}(s_\mathrm{c}))$.

Establishing conditions 1-3 guarantees that, whenever the code segment e_c executes in the code, there is an enabled event e_a in the TLS that causes a transition from the abstract image s_a under α of the concrete prestate s_c at the code level into an abstract state $e_\mathrm{a}(s_\mathrm{a})$ that is the abstract image under α of the concrete poststate $e_\mathrm{c}(s_\mathrm{c})$ at the code level. More concisely, conditions 1, 2, and 3 imply that there exists an abstract transition that models the concrete transition.

The relation of Event Code segments to abstract events can be slightly more complex than shown in Fig. 2. For example, in some cases, e_c may implement more than one event. However, these more complex cases can usually be handled similarly. When a concrete event implements n abstract events, for example, one looks for a partition $\mathrm{Pre_c} \equiv \mathrm{Pre}_\mathrm{c}^1 \oplus \ldots \oplus \mathrm{Pre}_\mathrm{c}^n$ of the concrete precondition $\mathrm{Pre_c}$ such that, when the ith part $\mathrm{Pre}_\mathrm{c}^i$ holds, the code e_c implements the ith abstract event. Then, one establishes, for each i, a commutative diagram analogous to the diagram in Fig. 2.

The argument that the kernel code of ED ensures data separation is based on relating executions of the code to executions in the TLS. To begin, we observe that α maps ED's initial state via α to an allowed initial state in the TLS. To support the remainder of the argument, the Event Code set $\mathrm{E_c}$ and the code-level map α are extended to cover the Other Code. Most Event Code segments consist of a single program statement. In contrast, Other Code contains many lengthy code segments which simply manipulate local variables inside a function or procedure and do not map to any abstract event. Such segments typically occur prior to an Event Code segment. We model these Other Code segments at the abstract level by a `no-op` ("do nothing") event implicitly included in the TLS. It is possible to map the effect of a segment of the Other Code to a `no-op` in the TLS because, unlike Event and Trusted Code, the Other Code has no access to MAIs. Because every code segment in the Event or Other Code is modeled either by an abstract TLS event with concrete and abstract transitions related as in Fig. 2 or by a `no-op` in the TLS, it follows that every execution of this part of the code corresponds to an execution in the TLS.

Trusted Code in the ED kernel can be related to the TLS as follows: First, it is established that no segment of the Trusted Code causes insecure data flows. Some segments of the Trusted Code have been verified, and the remaining segments have been certified externally to cause no insecure information flows. The state change caused by each Trusted Code segment is then shown to map to the result of either a `no-op` in the TLS or some sequence of events in the TLS. In the overall argument that an execution of concrete code always maps to a possible execution in the TLS, each Trusted Code segment is treated as an indivisible unit. In ED, this is possible because each Trusted Code segment

executes within a single partition and executions within a partition are never interrupted.

Combining this reasoning with the additional assurance that α relates concrete data and buffer memory areas to abstract ones and thus models all information flows involving MAIs, it follows that all kernel behavior relevant to data separation at the concrete level is modeled at the abstract level. Thus, the Data Separation Property proven at the abstract level also holds at the concrete level.

4.2 Uses and Proof Methods for Refinement

Although some details of how they are applied may vary, commutative diagrams are widely used to describe the required relationships between transitions at the concrete and abstract levels in a refinement relation (sometimes referred to as an *abstraction* relation).

When model checking is used to verify systems, a typical approach is to generate an abstract model automatically using data abstraction [20] or data type reduction [21] in a way that guarantees that the original system is a refinement of the model. Thus, any properties verified of the abstract model that are preserved under refinement will also hold for the system. In this approach, refinement is a given and need not be proved. For us, it was not feasible to use model checking to produce an abstract model. Due to the state explosion problem, model checking for verification has mostly been applied to hardware systems. Although, to some extent, methods such as abstraction refinement [22] have made it more feasible to apply model checking to software systems, model checking is better for detecting software bugs than for verifying software.

The concept of refinement also arises in the context of proving an implementation relation from a more concrete system model to a more abstract one. For example, Burch and Dill [23] use decision procedures and Cyrluk [24] uses PVS to verify that concrete models implement specifications by proving that a set of diagrams commute, where the diagram for each transition captures the correlation between sequences of instructions at the concrete and abstract levels. McMillan [21] avoids this use of commutative diagrams by using compositional model checking for proving implementation, in particular by model checking individual transitions separately and then proving that the results compose. In the context of hierarchical verification, Robinson and Levitt [25] use a diagram that relates concrete states to abstract states and concrete programs (or program fragments) to abstract transitions in which the poststate is mathematically defined in terms of the prestate. The abstraction mappings of Robinson and Levitt, as well as those of Burch and Dill and of Cyrluk, differ from ours in that they require the mapping to be a surjection, whereas we instead require it to be total. These authors' condition for implementation correctness is, like ours, that the diagram must commute. The surjection-versus-total difference reflects our different use of abstraction: They wish to prove that a more concrete program fully implements its abstraction, whereas we wish to deduce the correctness of the concrete program from the correctness of the abstraction.

We also add a method for proving that a program (or program fragment) implements an abstract transition by proving certain relationships among preconditions and postconditions at the concrete and abstract levels. Fig. 2 shows these required relationships, along with the commutative diagram. We also make explicit that 1) at the state variable level, the "state variables" mapped by the mapping function can be derived variables or logical variables that, for example, capture history and 2) postconditions are actually predicates on two states. Thus, we are able to handle state and transition information for which Abadi and Lamport [10] use history and prophecy variables.

5 Discussion

5.1 Applying Our Techniques to Other Security Properties

This section considers the class of security properties that can be verified using the techniques and formalization described in Sections 3 and 4. Two important classes of security properties are safety and liveness. In [26], Alpern and Schneider formally define *safety* and *liveness*, concluding that any property p can be expressed as the intersection of a safety property and a liveness property. Informally, a safety property states that nothing "bad" happens during execution and a liveness property states that something "good" happens during execution [27]. For Alpern and Schneider, a set of executions is called a *property* if membership in the set is determined by each execution alone, without reference to other executions in the set. McLean has identified a serious limitation of this notion of property—not every security property of interest is a property of executions [28]—and presents Noninterference as an example [29] of a property of sets of executions rather than a property of executions.

For the techniques presented in Section 3 to apply, a security property p must be preserved under refinement. It is well known that safety properties are preserved under refinement but that liveness properties are not [10]. Moreover, McLean has shown that properties such as Noninterference are not preserved by refinement [30]. Hence, our techniques can be used to guarantee security properties that are safety properties. It is easy to show that four of the security properties defined in Section 3.2—No-Exfiltration, Temporal Separation, Separation of Control, and Kernel Integrity—are safety properties. Therefore, all four properties are preserved by refinement. The fifth property, No-Infiltration, is not a safety property because it is not a property of executions but a property of sets of executions. However, it is easily shown to be preserved under refinement.

Our approach may be applied to many applications that, like ED, enforce access control. Applications that enforce access control restrict the operations that subjects (for example, users) can perform on objects (for example, data). As long as the access control policy can be represented as a safety property, our approach applies. A second important class of applications to which our approach applies are those described by Schneider, which use Execution Monitoring (EM) to enforce security [31]. Examples of EM mechanisms are reference monitors, firewalls, and other operating system and hardware-based enforcement mechanisms described in the literature. Excluded from this class are applications that use more information than would be available from observing only the states of a single system execution.

Schneider shows that the security properties enforced by EM mechanisms are safety properties.

5.2 Applying Our Method to Additional Kernel Properties

In ED's certification, our task was to develop a TLS of ED's kernel code, to verify that the TLS satisfies data separation, and, finally, to demonstrate conformance of the kernel code to the TLS. An important aspect of our approach is that, if required, we can construct a refinement of the TLS by adding new variables and events to the TLS to capture some behavior of (that is, events in) the Other Code. If the security properties that we wish to prove about this additional behavior are preserved by refinement, then we can formally state and prove the new security properties for the refinement of the TLS and show correspondence between the related portion of the Other Code and the new behavior. Because our proof method can be iterated through a series of refinements (see Appendix A), the proof of data separation remains valid under such a refinement of the TLS.

6 Lessons Learned

6.1 Software Design Decisions

Three software design decisions were critical in making code verification feasible. One major decision was to use a separation kernel, a single software module to mediate all memory accesses. A design that distributed the checking of memory accesses would have made the task of proving data separation much more difficult. A second critical decision was to keep the software simple. For example, once initiated, data processing in a partition was run to completion unless an exception occurred. In addition, ED's services were limited to the essential ones: The temptation to add new services late in the development was resisted. The third critical decision was enforcing "least privilege." For example, if a process only requires read access to a memory area, the kernel only grants read, *not* read and write, access.

6.2 Top-Level Specification

One significant challenge was to understand the externally visible security-relevant behavior of the separation kernel. Both scenarios and the SCR (Software Cost Reduction) tools [32], [33] were useful in extending our understanding of the kernel behavior. To begin, we formulated several scenarios, that is, sequences of events, and specified the kernel response to those events. After specifying a state machine model of the kernel in SCR, we ran the scenarios through the SCR simulator. As expected, formulating the scenarios and running them through the simulator exposed gaps in our understanding. Both the scenarios and the questions raised were valuable in eliciting details of the security-relevant kernel behavior from ED's development team.

Once the kernel's required behavior was understood, approximately 2.5 weeks were needed to formulate the TLS and the data separation property. The complete statement of the TLS, including the assumptions, is only 15 pages long. Keeping the size of the TLS small was critical for many reasons. It simplified communication with the other stakeholders, changing the specification when the kernel behavior changed, translating the specification into TAME, and proving that the TLS enforced data separation.

During the certification process, the natural language representation of the TLS enabled stakeholders with differing backgrounds and objectives—for example, the project manager and the evaluators—to communicate easily with the formal methods team about the kernel's required behavior. Discussion among the various stakeholders helped ensure that misunderstandings were avoided and issues were resolved early in the certification process. This natural language representation of the TLS for ED contrasts with the representations used in many other formal specifications of secure systems, which are often expressed in specialized languages such as ACL2 (for example, see [34]). Moreover, any ambiguity inherent in the natural language representation was removed by translating the TLS into TAME since the state machine semantics underlying TAME is expressed as a PVS theory. One component of the TLS in particular, the access control matrix, facilitated communication between the formal methods team and other stakeholders. Although the matrix was largely redundant of other parts of the TLS, stakeholders could easily understand the matrix and thus validate constraints on the access privileges of processes invoked by each event. The matrix was also useful in identifying the events and MAIs to be included in the TLS.

6.3 Mechanized Verification

TAME's specification and proof support significantly simplified the verification effort and required a total of about 3.5 weeks. Approximately 1.5 weeks was required to produce the final TAME model of the TLS and to document the correspondence between the TAME model and the TLS. Some of this time was required to choose appropriate data structures for representing the state variables and the parameters of actions in TAME. The higher order nature of PVS made it feasible to handle the unspecified number of memory areas in the TLS by representing the overall memory content in TAME as a function from a set of memory areas to storable values and, in general, to produce a very compact TAME specification (368 lines long). Once the data representations were chosen, translating the TLS and the five subproperties into TAME required only three days. Adjusting the TAME specification to reflect later changes in the TLS required only a few hours. To illustrate the TAME representation, Appendix B provides TAME versions of three of the five subproperties.

About two weeks was needed to formally verify that the TLS enforces data separation. Most of this time was spent formulating an efficient proof approach and then developing a new TAME strategy to implement the approach. The new PVS strategy, designed to simplify the proof guidance in the presence of the data structures used in the TAME specification, was used in the proofs of all subproperties and has subsequently proven useful in other TAME applications. Once the strategy was developed, the time required to develop the proof scripts interactively in TAME was one day. Adding and proving a new subproperty suggested by an evaluator required under one hour. The proof script of each subproperty executes in two minutes or less.

6.4 Showing Code Conformance

For one month, we experimented with several different approaches for demonstrating conformance between the TLS and the annotated C code before the approach presented in Section 3.5 was selected. Once an approach was selected, a total of about five weeks was required to establish conformance. The formal foundation for the correspondence argument required one week. Three weeks were needed to construct the correspondence of Event Code to TLS events, that is, developing the code-level assertions necessary for the TLS preconditions and postconditions to hold and locating the corresponding event code and assertions in the annotated C code. One day was spent using the Xcode tool to locate all calls to functions that can reset the MMU and manually verifying that the permissions for the Other Code did not include access to MAIs. One week was needed to add the required assertions to the annotated code. Our method for demonstrating code conformance in the ED kernel relies on the notions of MAIs and Event Code. The extent to which our method can be extended to other applications depends on whether an analogous method of identifying the Event Code (and the Trusted Code) can be found. In addition, as noted in Section 5.1, in our method, the properties to be proven must be preserved under refinement. Both the EM class of applications and the applications that enforce access control policies are likely to meet both conditions.

7 Open Problems

7.1 Code Annotations

For many years, some researchers have recommended annotating code with preconditions, postconditions, and invariants (for example, see [35], [36]). Such code annotations are already used in practice. For example, developers at Praxis annotate SPARK programs with assertions and use tools to automatically check the assertions [37]. Furthermore, in the largest Microsoft product groups, annotations are a mandated part of the software development process [38]. However, manual annotation of source code remains rare in the wider software development industry because it is highly labor intensive [39]. Although tools for checking code annotations would be valuable, tools that can construct preconditions and postconditions automatically are even more valuable. Some promising initial research on the automatic extraction of a specification from code has been published (for example, see [40]). However, to date, current research has mostly focused on extracting specifications to detect code vulnerabilities. Extracting assertions from code for checking security properties such as access privileges and other assertions, like those shown in Fig. 1, has not been investigated.

7.2 Automatic Generation of Test Cases from Assertions

Recently, many new techniques for specification-based testing have been proposed. Such techniques (see, for example, [41]) construct a set of test cases for checking a software implementation against a formal requirements specification. One promising new direction would be to construct test cases from preconditions and postconditions that annotate the source code and then use the test cases to check that a given program (such as a program that includes the annotated C code in Fig. 1) satisfies the asserted preconditions and postconditions. Validating the C code in this manner would provide high assurance of both the security and functional correctness of the C code. Reference [42] describes an approach that uses automatic test case generation from preconditions and postconditions to find bugs in Java code.

7.3 A Code Conformance Proof Assistant

The semantic distance between the abstract TLS required for a Common Criteria evaluation and a low-level C program is huge. Although the TLS describes the security-relevant program behavior in terms of sets, functions, and relations, the description of the behavior of a C program is in terms of low-level constructs such as arrays, integers, and bits stored in registers and memory areas. Hence, an automatic demonstration of conformance of low-level C code to a TLS is unrealistic. A more realistic goal is a proof assistant with two inputs—a C program annotated with assertions and a TLS of the security-relevant functions of that program—for helping the user establish that the C program satisfies the TLS.

7.4 Automatic Code Generation

One promising way of obtaining high assurance that an implementation satisfies a set of critical security properties is to generate code automatically from a specification that has been proven to satisfy the properties. Automatic code generation is already feasible for some low-level specification languages such as Esterel [43]. Although constructing efficient source code from more abstract specifications is possible for simple program constructs using simple data types (for example, see [44]), new research is needed to produce efficient code from specifications containing richer constructs and data types. Such technology should drastically reduce the effort required to produce efficient code and to increase assurance that the code satisfies critical security properties.

8 Related Work

In the 1980s, the SCOMP [45], SeaView [46], LOCK [47], and Multinet Gateway [48] projects all applied formal methods to the specification and verification of systems. All developed TLSs and formal statements of the system security policies. For SCOMP, Multinet Gateway, and LOCK, the TLS was formally shown to satisfy the security policy. For SeaView, only two of the 31 operations in the TLS were verified against the security policy model [6]. Conformance between the TLS and the SCOMP code was shown by constructing several mappings: the English language to the TLS, the TLS to pseudocode, and the TLS to actual code [5]. The mapping was top down from the TLS to the code. As a result, some code was unmapped. This approach is similar to our mapping of Event Code to the TLS, although the mapping is in the other direction. The LOCK project constructed mappings partially relating the TLS to the source code. Specification-based testing provided additional evidence of correspondence. In Multinet Gateway, verification conditions were generated to show conformance between the specification and the code. If and how these conditions were discharged is unclear. Each project used tools to aid in specification and verification: SCOMP used HDM [49],

Seaview used EHDM [50], and Multinet Gateway and LOCK used Gypsy [51]. More recently, in 2006, we formulated a second possible approach to software verification based on TAME, which uses verified formal pseudocode as "glue" relating a TLS to actual code [52].

In [34], [53], Greve, Wilding, and Vanfleet (GWV) present an ACL2 model for a generic separation kernel. In the model, a function describes the possible information flows between memory areas. This notion of flow is not as fine-grained as our model, where access (with its possible information flows) is granted to each process only when it executes in a partition, thus providing least privilege in addition to separation. In the GWV approach, separation includes No-Exfiltration and No-Infiltration but not Temporal Separation since the model does not allow reconfigurable partitions. How the GWV model was used to verify the AAMP7 microprocessor is described in [54], [55]. A traditional verification process was followed: Build a formal security policy, both an abstract and a detailed model, and an implementation, then prove that the abstract model satisfies the security policy and show correspondence between the abstract and detailed models and between the detailed model and the implementation. Whether correctness was proven at either the detailed design level or the code level is unclear.

9 Conclusions

This paper has introduced a novel and affordable approach for verifying security down to the source code level. The approach begins with a well-defined security property, builds the minimal state machine model needed to prove that the model satisfies the property, and proves, using a mechanical verifier, that the security model satisfies the property. Once complete, the code is annotated with preconditions and postconditions and is then partitioned into Event, Trusted, and Other Code. The final step is to 1) demonstrate conformance of the Event Code and the code preconditions and postconditions with the internal events and preconditions and postconditions of the TLS and 2) show that the Trusted Code and the Other Code are benign. Tools such as model checkers and theorem provers are already available for verifying that a formal specification satisfies a security property of interest. A research challenge is to develop tools for the following:

1. validating and constructing preconditions and postconditions from the source code, including the C code,
2. automatically generating test cases that check C code annotations,
3. showing conformance of annotated code with a TLS, and
4. automatically constructing efficient provably correct code from specifications.

Research that addresses these four problems should significantly increase the affordability of constructing verified security-critical software.

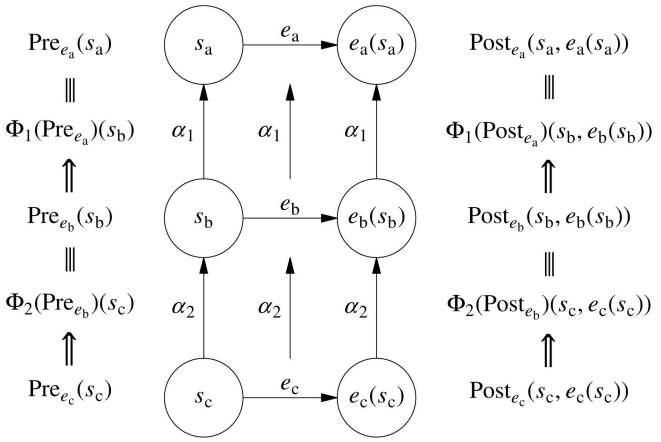

Fig. 3. Relationships in successive refinements.

Appendix A
Iterating Our Refinement Method

Fig. 3 illustrates the mappings, predicates, and relationships between assertions connected with the proof of successive refinements from an automaton at level c through an automaton at level b to an automaton at level a. We wish to prove that if the analogs of conditions 1, 2, and 3 from Section 4.1 hold for the c-to-b and b-to-a relations, then conditions 1, 2, and 3 hold for the composed c-to-a relation in which $\alpha \triangleq \alpha_1 \circ \alpha_2$ and $\Phi \triangleq \Phi_2 \circ \Phi_1$. In analogy with the notation in Section 4, we use s_b to denote $\alpha_2(s_c)$, s_a to denote $\alpha_1(s_b)$, and S_a, S_b, and S_c to denote the sets of states at levels a, b, and c. We first need a lemma.

Lemma A.1. *Let $\alpha : S_c \to S_a$ and let Φ be the map from predicates on S_a to predicates on S_c induced by α, that is, such that, for any predicate $\mathrm{P_a}$ and any element $s_c \in S_c$, $\Phi(\mathrm{P_a})(s_c) \triangleq \mathrm{P_a}(\alpha(s_c))$. If $\mathrm{P_a}$ and $\mathrm{Q_a}$ are predicates on S_a such that $\mathrm{P_a} \Rightarrow \mathrm{Q_a}$, then $\Phi(\mathrm{P_a}) \Rightarrow \Phi(\mathrm{Q_a})$.*

Proof. Suppose that $\mathrm{P_a}$ and $\mathrm{Q_a}$ are two predicates on S_a, for which $\mathrm{P_a} \Rightarrow \mathrm{Q_a}$. This means that, $\forall s_a \in S_a$, $\mathrm{P_a}(s_a) \Rightarrow \mathrm{Q_a}(s_a)$. Let s_c be any element of S_c. Then,

$$\begin{aligned} \Phi(\mathrm{P_a})(s_c) &= \mathrm{P_a}(\alpha(s_c)) && \text{(by the definition of } \Phi) \\ &\Rightarrow \mathrm{Q_a}(\alpha(s_c)) && \text{(since } \mathrm{P_a} \Rightarrow \mathrm{Q_a}) \\ &= \Phi(\mathrm{Q_a})(s_c) && \text{(by the definition of } \Phi). \end{aligned}$$

□

Next, we define the notion of an *annotated transition*.

Definition A.1. *Let S be a set of states and let $E \subset S \times S$ be a set of transitions on S. An annotated transition is a transition $e \in E$ accompanied by a one-state predicate Pre_e on S and a two-state predicate Post_e on S.*

Now, we can state the theorem formally:

Theorem A.2. *Let **A**, **B**, and **C** be automata with state spaces S_b, S_b, and S_c and sets of annotated transitions E_a, E_b, and E_c, respectively. Let $\alpha_2 : S_c \to S_b$ and $E_c \to E_b$ and $\alpha_1 : S_b \to S_a$ and $E_b \to E_a$ be refinement mappings, that is, mappings that, together with their induced mappings Φ_2 and Φ_1 on predicates and the transition annotations, satisfy the*

appropriate analogs of conditions 1, 2, and 3 from Section 4.1. For convenience, we refer to these conditions as conditions $1_{b,c}$, $2_{b,c}$, *and* $3_{b,c}$ *and conditions* $1_{a,b}$, $2_{a,b}$, *and* $3_{a,b}$. *Then if* $\alpha \triangleq \alpha_1 \circ \alpha_2$ *and* $\Phi \triangleq \Phi_2 \circ \Phi_1$, *the mappings* α *and* Φ *satisfy conditions 1, 2, and 3, and hence,* $\alpha : S_c \to S_a$, *and* $E_b \to E_a$ *is a refinement mapping.*

Proof. Suppose that the hypotheses of Theorem A.2 hold. Then, we need to establish that conditions 1, 2, and 3 hold.

For condition 1, we can argue as follows:

(i) $\mathrm{Pre}_{e_c} \Rightarrow \Phi_2(\mathrm{Pre}_{e_b})$ (by condition $1_{b,c}$)
(ii) $\mathrm{Pre}_{e_b} \Rightarrow \Phi_1(\mathrm{Pre}_{e_a})$ (by condition $1_{a,b}$)
(iii) $\Phi_2(\mathrm{Pre}_{e_b}) \Rightarrow \Phi_2(\Phi_1(\mathrm{Pre}_{e_a}))$ (by (ii) and Lemma A.1)

and, therefore,

(iv) $\mathrm{Pre}_{e_c} \Rightarrow \Phi_2(\Phi_1(\mathrm{Pre}_{e_a}))$ (by (i) and (iii))
(v) $\mathrm{Pre}_{e_c} \Rightarrow \Phi(\mathrm{Pre}_{e_a})$ (by the definition of Φ)

For condition 2, first note that the first part of condition 2, which relates Pre_{e_c} to Post_{e_c}, follows from the first part of condition $2_{b,c}$. The remainder of the argument, which relates Post_{e_c} to Post_{e_a}, is totally analogous to that for condition 1.

To prove condition 3, we note that if $\mathrm{Pre}_{e_c}(s_c)$ holds, then by condition $3_{b,c}$, the lower square in Fig. 3 commutes. Furthermore, we have

$$\begin{array}{ll} & \mathrm{Pre}_{e_c}(s_c) \\ \Rightarrow & \Phi_2(\mathrm{Pre}_{e_b})(s_c) \quad (\text{by condition } 1_{b,c}) \\ \equiv & \mathrm{Pre}_{e_b}(s_b) \quad (\text{by definition of } \Phi_2, \text{ since } s_c = \alpha_2(s_b)) \end{array}$$

and hence $\mathrm{Pre}_{e_b}(s_b)$ holds. By condition $3_{a,b}$, this implies that the upper square commutes. Therefore, the diagram as a whole commutes and we have

$$e_a \circ \alpha_1 \circ \alpha_2 = \alpha_1 \circ \alpha_2 \circ e_c.$$

By the definition of α, this means that

$$e_a \circ \alpha = \alpha \circ e_c,$$

and we are done. □

APPENDIX B
TAME REPRESENTATION OF SEPARATION

To provide some details of the TAME representation of ED, we show how three of the five subproperties of the separation property verified for ED, Temporal Separation, No-Exfiltration, and No-Infiltration, are represented in TAME. For each subproperty, we first repeat its natural language representation from Section 3.2 and then show and explain its TAME representation.

B.1 Temporal Separation

Natural language version

(Temporal Separation) For all states s in S, for all i, $1 \le i \le n$, if the partition id c_s is 0, then the k data areas of partition i are clear, that is, $D^1_{i,s} = 0, \ldots, D^k_{i,s} = 0$.

TAME version

```
Inv_ClearPart(s:states):bool =
  (FORALL (i:PartIndex): (NONE?(PartId(s)) =>
    (FORALL (n:DataAreaIndex):
      Clear?(MemContent(DataArea(i,n),s)))));

lemma_ClearPart: LEMMA (FORALL (s:states):
   reachable(s) => Inv_ClearPart(s));
```

The TAME representation of the Temporal Separation property is the state-invariant lemma `lemma_ClearPart`, which states that the invariant `Inv_ClearPart` holds for every reachable state `s`. In the invariant `Inv_ClearPart`, `PartIndex` and `DataAreaIndex` are the types of partition indices and data area indices, defined simply to be nonempty, uninterpreted types. Thus, there can be an arbitrary nonzero number of partitions, each with the same but arbitrary nonzero number of data areas. `PartId(s)` represents c_s, the current partition id in the current state. `NONE?(PartId(s))` is true when c_s is 0, that is, exactly when no partition processing is taking place. `MemContent` is a function that maps a memory area and a state to the memory content of that memory area in that state. Finally, the predicate `Clear?` is true of the memory content of a data area when that data area is clear.

B.2 No-Exfiltration

Natural language version

(No-Exfiltration) Suppose that states s and s' are in state set S, event e is in H, memory area a is in $\mathcal{M}$, and j is a partition, $1 \le j \le n$. Suppose further that $s' = T(e, s)$. If e is an event in $P_j \cup E^{\mathrm{In}}_j \cup E^{\mathrm{Out}}_j$ and $a_s \neq a_{s'}$, then a is in A_j.

TAME version

```
No_Exfiltration: LEMMA
 (FORALL (E:actions, s:states, m:MemAreas,
          j:PartIndex):
    (enabled(E,s) & Isin(m,PartMemAreas(j)) &
     (NONE?(PartId(s)) OR
      (Part?(PartId(s)) & NOT(Id(PartId(s))=j))))
 => ((InBuff?(E) & InBuff_Index(E)=j) OR
     (OutBuff?(E) & OutBuff_Index(E)=j) OR
     MemContent(m,s)=MemContent(m,trans(E,s))));
```

The TAME version `No_Exfiltration` of the No-Exfiltration property corresponds to the contrapositive of the natural language version. In the TAME representation, the event e is represented by an action E. The state s is represented by `s` and the state s' is represented by `trans(E,s)`, that is, the result of a transition due to action `E` in state `s`. For the current partition id `PartId(s)` in state `s`, either `NONE?` holds, that is, no partition processing is occurring, or `Part?` holds, in which case, partition processing is occurring in partition `id(PartId(s))`. The assertion `enabled(E,s)` means that the precondition of action `E` holds in state `s`. When `E` is an internal action, this precondition ensures that `E` is an internal action for Partition `PartId(s)`. The condition `InBuff?(E)&InBuff_Index(E)` = `j` is true when `E` fills the input buffer of Partition `j`. The analogous condition with `Out` in place of `In` is true when `E` empties the output buffer of Partition `j`. These parts of the conclusion of property `No_Exfiltration` cover the cases when action `E` is an external event for Partition `j`. Thus, property

No_Exfiltration says that, if m is a memory area in Partition j and E either is an external action or is an internal action in some partition other than Partition j, then either E is an external action for Partition j or E does not change the content of m.

B.3 No-Infiltration

Natural language version

(No-Infiltration) Suppose that states s_1, s_2, s_1', and s_2' are in S, event e is in H, and i is a partition, $1 \leq i \leq n$. Suppose further that $s_1' = T(e, s_1)$ and $s_2' = T(e, s_2)$. If, for all a in A_i, $a_{s_1} = a_{s_2}$, then, for all a in A_i, $a_{s_1'} = a_{s_2'}$.

TAME version

```
No_Infiltration: LEMMA
 (FORALL (E:actions, s1,s2:states, m:MemAreas,
          i:PartIndex):
    enabled(E,s1) & enabled(E,s2) &
    Part?(PartId(s1)) & Id(PartId(s1))=i &
    Part?(PartId(s2)) & Id(PartId(s2))=i &
    Isin(m, PartMemAreas(i)) &
    (FORALL (m1:MemAreas):Isin(m1,PartMemAreas(i))
      => MemContent(m1,s1)=MemContent(m1,s2))
 => MemContent(m,trans(E,s1))
      = MemContent(m,trans(E,s2)));
```

The preceding explanation of the notation in lemma_ClearPart and No_Exfiltration should make it clear that the TAME version No_Infiltration of the No-Infiltration Property is equivalent to the natural language version.

ACKNOWLEDGMENTS

The authors acknowledge Gerard Allwein of NRL for telling them about the Xcode tool. They also acknowledge the monumental effort of the logician who annotated the kernel code with preconditions and postconditions and of the ED project leader, who had the foresight to include a separation kernel and keep the design simple. Without the annotated code and solid design decisions, the authors would have been unable to obtain the results described in this paper. The authors also thank the members of the ED design team for answering questions about ED's operational behavior. This research was supported by the US Office of Naval Research. This paper is an extended version of a paper presented at the 13th ACM Conference on Computer and Communications Security (CCS '06) held in Alexandria, Virginia, 30 October-3 November 2006 [56].

REFERENCES

[1] J.S. Moore, T.W. Lynch, and M. Kaufmann, "A Mechanically Checked Proof of the AMD5K86TM Floating-Point Division Program," *IEEE Trans. Computers,* vol. 47, no. 9, Sept. 1998.

[2] J. Rushby, "A Formally Verified Algorithm for Clock Synchronization under a Hybrid Fault Model," *Proc. 13th ACM Symp. Principles of Distributed Computing,* Aug. 1994.

[3] C. Meadows, "Analysis of the Internet Key Exchange Protocol Using the NRL Protocol Analyzer," *Proc. IEEE Symp. Security and Privacy,* 1999.

[4] J. Juerjens, "Sound Methods and Effective Tools for Model-Based Security Engineering with UML," *Proc. 27th Int'l Conf. Software Eng.,* 2005.

[5] T. Benzel, "Analysis of a Kernel Verification," *Proc. IEEE Symp. Security and Privacy,* Apr. 1984.

[6] R. Whitehurst and T. Lunt, "The SeaView Verification," *Proc. Second IEEE Computer Security Foundations Workshop,* June 1989.

[7] J. Rushby, "Design and Verification of Secure Systems," *Proc. Eighth ACM Symp. Operating System Principles,* Dec. 1981.

[8] B. Lampson, "Protection," *Proc. Fifth Princeton Conf. Information Sciences and Systems,* Mar. 1991.

[9] N. Shankar, S. Owre, J.M. Rushby, and D.W.J. Stringer-Calvert, "PVS Prover Guide Version 2.4," technical report, Computer Science Laboratory, SRI Int'l, Nov. 2001.

[10] M. Abadi and L. Lamport, "The Existence of Refinement Mappings," *Theoretical Computer Science,* vol. 82, no. 2, pp. 253-284, 1991.

[11] C. Adams, "Keeping Secrets in Integrated Avionics," *Aviation Today,* 2004.

[12] J. Anderson, "Computer Security Technology Planning Study," Technical Report ESD-TR-73-51, Hanscom AFB, ESD/AFSC, 1972.

[13] "Common Criteria for Information Technology Security Evaluation,"Parts 1-3: Technical Reports CCIMB-2004-01-001 through CCIMB-2004-01-003, Version 2.2, Revision 256, Jan. 2004.

[14] C.E. Landwehr, C.L. Heitmeyer, and J.D. McLean, "A Security Model for Military Message Systems," *ACM Trans. Computer Systems,* vol. 2, no. 3, pp. 198-222, 1984.

[15] J. McLean, C. Landwehr, and C. Heitmeyer, "A Formal Statement of the Military Message System Security Model," *Proc. IEEE Symp. Security and Privacy,* pp. 188-194, 1984.

[16] M. Archer, "TAME: Using PVS Strategies for Special-Purpose Theorem Proving," *Annals of Math. and Artificial Intelligence,* vol. 29, nos. 1-4, pp. 139-181, 2000.

[17] M. Archer, C.L. Heitmeyer, and E. Riccobene, "Proving Invariants of I/O Automata with TAME," *Automated Software Eng.,* vol. 9, no. 3, pp. 201-232, 2002.

[18] S. Owre, J. Rushby, N. Shankar, and F. von Henke, "Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS," *IEEE Trans. Software Eng.,* vol. 21, no. 2, pp. 107-125, Feb. 1995.

[19] Xcode Version 2.1, http://developer.apple.com/tools/xcode/index.html, 2007.

[20] E.M. Clarke, O. Grumberg, and D.E. Long, "Model Checking and Abstraction," *Proc. 21st ACM Symp. Principles of Programming Language,* 1994.

[21] K.L. McMillan, "Verification of Infinite State Systems by Compositional Model Checking," *Proc. 10th IFIP WG 10.5 Advanced Research Working Conf. Correct Hardware Design and Verification Methods,* 1999.

[22] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-Guided Abstraction Refinement," *Proc. 12th Int'l Conf. Computer-Aided Verification,* 2000.

[23] J.R. Burch and D.L. Dill, "Automatic Verification of Pipelined Microprocessors Control," *Proc. Sixth Int'l Conf. Computer-Aided Verification,* vol. 818, pp. 68-80, 1994.

[24] D. Cyrluk, "Microprocessor Verification in PVS: A Methodology and Simple Example," Technical Report SRI-CSL-93-12, 1993.

[25] L. Robinson and K.N. Levitt, "Proof Techniques for Hierarchically Structured Programs," *Comm. ACM,* vol. 20, no. 4, pp. 271-283, 1977.

[26] B. Alpern and F.B. Schneider, "Defining Liveness," *Information Processing Letters,* vol. 21, no. 4, pp. 181-185, 1985.

[27] L. Lamport, "Proving the Correctness of Multiprocess Programs," *IEEE Trans. Software Eng.,* vol. 3, no. 2, pp. 125-143, 1977.

[28] J. McLean, "A General Theory of Composition for Trace Sets Closed under Selective Interleaving Functions," *Proc. IEEE Symp. Research in Security and Privacy '94,* pp. 79-93, May 1994.

[29] J.A. Goguen and J. Meseguer, "Security Policies and Security Models," *Proc. IEEE Symp. Research in Security and Privacy '92,* pp. 11-20, May 1992.

[30] J. McLean, "Security Models," *Encyclopedia of Software Eng.,* J. Marciniak, ed., John Wiley & Sons, 1994.

[31] F.B. Schneider, "Enforceable Security Policies," *ACM Trans. Information and System Security,* vol. 3, no. 1, pp. 30-50, 2000.

[32] C.L. Heitmeyer, R.D. Jeffords, and B.G. Labaw, "Automated Consistency Checking of Requirements Specifications," *ACM Trans. Software Eng. and Methodology,* vol. 5, no. 3, pp. 231-261, 1996.

[33] C.L. Heitmeyer, M. Archer, R. Bharadwaj, and R.D. Jeffords, "Tools for Constructing Requirements Specifications: The SCR Toolset at the Age of Ten," *Computer Systems: Science and Eng.,* vol. 20, no. 1, 2005.

[34] D. Greve, M. Wilding, and W.M. Vanfleet, "A Separation Kernel Formal Security Policy," *Proc. Fourth Int'l Workshop ACL2 Prover and Its Applications,* July 2003.
[35] B. Meyer, "Applying 'Design by Contract'," *Computer*, vol. 25, no. 10, pp. 40-51, Oct. 1992.
[36] M. Chechik and J.D. Gannon, "Automatic Analysis of Consistency between Requirements and Designs," *IEEE Trans. Software Eng.*, vol. 27, no. 7, pp. 651-672, 2001.
[37] J. Barnes, *High Integrity Software: The SPARK Approach to Safety and Security.* Addison-Wesley, 2003.
[38] M. Das, "Formal Specifications on Industrial-Strength Code: From Myth to Reality," *Proc. 18th Int'l Conf. Computer-Aided Verification,* Aug. 2006.
[39] S. Hallem, B. Chelf, Y. Xie, and D.R. Engler, "A System and Language for Building System-Specific, Static Analyses," *Proc. ACM Conf. Programming Language Design and Implementation,* pp. 69-82, 2002.
[40] T. Kremenek, P. Twohey, G. Back, A. Ng, and D. Engler, "From Uncertainty to Belief: Inferring the Specification Within," *Proc. Seventh Symp. Operating Systems Design and Implementation,* Dec. 2006.
[41] A. Gargantini and C.L. Heitmeyer, "Using Model Checking to Generate Tests from Requirements Specifications," *Proc. Seventh European Software Eng. Conf. and Seventh ACM Symp. Foundations of Software Eng.*, 1999.
[42] Y. Smaragdakis and C. Csallner, "Combining Static and Dynamic Reasoning for Bug Detection," *Proc. First Int'l Conf. Tests and Proofs,* pp. 1-16, 2007.
[43] SCADE Tool Suite, http://www.esterel-technologies.com/products/scade-suite, 2007.
[44] T. Rothamel, C. Heitmeyer, E. Leonard, and Y.A. Liu, "Generating Optimized Code from SCR Specifications," *Proc. ACM Conf. Languages, Compilers and Tools for Embedded Systems,* June 2006.
[45] L. Fraim, "Secure Office Management System: The First Commodity Application on a Trusted System," *Proc. Fall Joint Computer Conf. Exploring Technology: Today and Tomorrow,* 1987.
[46] T. Lunt, D. Denning, R. Schell, M. Heckman, and W. Shockley, "The SeaView Security Model," *IEEE Trans. Software Eng.*, vol. 16, no. 6, pp. 593-607, June 1990.
[47] R. Smith, "Cost Profile of a Highly Assured Secure Operating System," *ACM Trans. Information and System Security,* vol. 4, no. 1, Feb. 2001.
[48] S. Gerhart, D. Craigen, and T. Ralston, "Case Study: Multinet Gateway System," *IEEE Software,* pp. 37-39, 1994.
[49] L. Robinson, "The HDM Handbook, Vol. 1: The Foundations of HDM," SRI Project 4828, technical report, SRI Int'l, 1979.
[50] J. Rushby, F. von Henke, and S. Owre, "An Introduction to Formal Specification and Verification Using EHDM," Technical Report CSL-91-2, SRI Int'l, Feb. 1991.
[51] D. Good, "Mechanical Proofs about Computer Programs," *Math. Logic and Programming Languages,* C. Hoare and J. Shepherdson, eds., pp. 55-75, Prentice Hall, 1985.
[52] M. Archer and E. Leonard, "Establishing High Confidence in Code Implementations of Algorithms Using Formal Verification of Pseudo-Code," *Proc. Third Int'l Verification Workshop,* 2006.
[53] J. Alves-Foss and C. Taylor, "An Analysis of the GWV Security Policy," *Proc. Fifth Int'l Workshop ACL2 Prover and Its Applications,* 2004.
[54] D. Greve, R. Richards, and M. Wilding, "A Summary of Intrinsic Partitioning Verification," *Proc. Fifth Int'l Workshop ACL2 Prover and Its Applications,* 2004.
[55] R. Richards, D. Greve, M. Wilding, and W. Vanfleet, "The Common Criteria, Formal Methods and ACL2," *Proc. Fifth Int'l Workshop ACL2 Prover and Its Applications,* 2004.
[56] C. Heitmeyer, M. Archer, E. Leonard, and J. McLean, "Formal Specification and Verification of Data Separation in a Separation Kernel for an Embedded System," *Proc. 13th ACM Conf. Computer and Comm. Security,* Oct.-Nov. 2006.

Constance L. Heitmeyer heads the Software Engineering Section of the Naval Research Laboratory's Center for High Assurance Computer Systems. She is the chief designer of the Software Cost Reduction (SCR) toolset for formally specifying and analyzing requirements. Her research interests include software requirements, formal methods, real-time computing, and formal models for computer security. She was the program chair of the Third and Fourth ACM/IEEE Conference on Formal Methods and Models for Codesign (MEMOCODE 2005 and 2006). She has been a member of the editorial boards of the *ACM Transactions on Software Engineering and Methodology*, *Software and Systems Engineering*, *Innovations in Systems and Software Engineering*, and *Real-Time Systems*. She is also the formal methods subject area editor of *Requirements Engineering*. She is the author of more than 100 papers. Her papers can be found at http://chacs.nrl.navy.mil/personnel/heitmeyer.html. She is a member of the IEEE, the IEEE Computer Society, and the ACM.

Myla M. Archer received the AM degree in mathematics from Harvard University and the PhD degree in computer science from the University of Illinois at Urbana-Champaign. She has been a research computer scientist in the Software Engineering Section of the Naval Research Laboratory's (NRL's) Center for High Assurance Computer Systems since 1995. Prior to joining the NRL, she taught mathematics at Wheaton College, Norton, Massachusetts, and served on the Computer Science Faculty at the University of California, Davis. She was the general chair of the 1991 Tutorial and Workshop on the HOL Theorem Proving System and Its Applications, the doctoral consortium chair of the Third IEEE Symposium on Requirements Engineering (ISRE '97), and a coorganizer of the First International Workshop on Design and Application of Strategies/Tactics in Higher Order Logic (STRATA '03) and the Sixth International Workshop on Strategies in Automated Deduction (STRATEGIES '06). Her research interests include formal methods, verification, requirements, and strategies in automated deduction. She is a member of the IEEE Computer Society and the ACM.

Elizabeth I. Leonard received the MSE and PhD degrees in computer science from the Johns Hopkins University. She is a research computer scientist in the Software Engineering Section of the Naval Research Laboratory's (NRL's) Center for High Assurance Computer Systems. Prior to joining the NRL, she held a fellowship at the Space Telescope Science Institute and served as a research intern at the US Air Force Rome Laboratory. She served on the program committee and as the publications chair of the Third, Fourth, and Fifth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE '05, MEMOCODE '06, and MEMOCODE '07). Her research interests include formal methods, security policy analysis, security modeling, automatic code generation, and verification. She is a member of the IEEE Computer Society and the ACM.

John D. McLean is the superintendent of the Information Technology Division at the Naval Research Laboratory (NRL), where he was the director of the Division's Center for High Assurance Computer Systems and the senior scientist for Information Assurance. While with NRL, he has been a senior research fellow at the University of Cambridge's Centre for Communications Systems Research and an adjunct professor of computer science at the University of Maryland, the National Cryptologic School, and Troiseme Cycle Romand d'Informatique. He was an associate editor for *Distributed Computing*, the *Journal of Computer Security*, the *ACM Transactions on Information and System Security*, and the *International Journal of Information and Computer Security*. His research interests include formal methods and formal models for computer security.